ARI Research Note 87-63

World Modeler SEI Library Builder

Peter Shell
University of California, Irvine

for

Contracting Officer's Representative
Judith Orasanu

BASIC RESEARCH LABORATORY
Michael Kaplan, Director

U. S. Army

Research Institute for the Behavioral and Social Sciences

December 1987

U. S. ARMY RESEARCH INSTITUTE

FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the

Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON
Technical Director

WM. DARRYL HENDERSON
COL, IN
Commanding

Research accomplished under contract
for the Department of the Army

University of California, Irvine

Technical review by

Dan Ragland

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ARI Research Note 87-63	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* World Modeler SEI Library Builder		5. TYPE OF REPORT & PERIOD COVERED Interim Report July 86 - July 87
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Peter Shell		8. CONTRACT OR GRANT NUMBER(s) MDA903-85-C-0324
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of California, Irvine Department of Information and Computer Science Irvine, CA 92717		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2Q161102B74F
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Research Institute for the Behavioral and Social Sciences, 5001 Eisenhower Avenue, Alexandria, VA 22333-5600		12. REPORT DATE December 1987
		13. NUMBER OF PAGES 8
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)* - -		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE n/a

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

- -

18. SUPPLEMENTARY NOTES

Judith Orasanu, contracting officer's representative

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

World Modeler	Artificial Intelligence
SEI	Learning Models
LispSEI	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This research note describes two different libraries (sensor and effector), which provide an abstract data type called the sensor. The sensor libraries allow an organism program to receive environmental data in different forms and degrees of filtering, while the effector libraries allow the organism to adapt to the environment in a "fair" way.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

World Modeler SEI Library Builder

Peter Shell

1. Introduction

The SEI librarian provides an abstract data type called the sensor, which is used for perceiving the world. It also provides facilities for defining filtering functions over perceived things in the world. It runs in Commonlisp, and requires that LispSEI is loaded in first. Although anybody may write a library function, they should be defined through the filter-defining macros described in this document. Some filters of general use are already provided (see Appendix I).

There are two different kinds of libraries: sensor libraries and effector libraries. The sensor libraries allow the organism program to receive environmental data in various forms and with various degrees of filtering. They only provide information about the *physical environment*, and no information about the organism's internal state. The effector libraries allow the organism to make changes to the environment, while ensuring that the basic laws of conservation of energy and momentum are obeyed as much as is practical.

2. Sensor Libraries

The SEI library provides, as well as some basic filtering functions, an abstract data type called the *sensor*. A sensor is something attached to an object (usually part of the organism), and is associated with one of the sense types (vision, hearing, smell, taste, and touch). Each of these senses is capable of sensing a different kind of World Modeler *entity*, e.g., objects, emissions, etc., as described in the LispSEI manual. The senses and entities form the following correspondences:

Sense	**Entity**
Vision	Object
Hearing	Emissions of type *sound*.
Smell	Emissions of type *smell*.
Taste	Taste fields of objects.
Touch	Contact-type.

Various types of filtering functions may also be associated with sensors. Only entities which pass through each of these filters are sensed by the organism. The sensor library provides a handful of pre-defined filtering functions, as well as filter-defining macros.

(def-sensor <sensor-name> <sense-type> <object> <entity-filters> <field-mungers>)

Defines a sensor called *<sensor-name>*, which can sense the *<sense-type>* sense, and attaches it to *<object>*[1], making that object a *sensor*. <object> may be either the name of an object, in which case it is an atom, or it can be a list whose first element is the name of an object and whose remaining elements are coordinates specifying the location (in world-wide x, y and z coordinates) on the object where the sensor should be. If the location is not supplied, then the geometric center of the object is used.

The sensor is associated with the ordered list of filters and constraints named in *<entity-filters>* (see **def-filter** and **def-constraint**, below), and with the list of field mungers named in *<field-mungers>* (see **def-field-munger**, below). As with **<object>**, each of the entity filters and field mungers named in **<entity-filters>** and **<field-mungers>** may be either atoms or lists. If they are atoms, then they name a filter or munger which takes no arguments; if they are a list, then the car of the list names the filter or munger, and the rest of the list gives values for sensor parameters with respect to that filter or munger. *The number of parameters given to each filter and munger* must be the same as the number of parameters declared by **def-filter**, **def-constraint** or **def-field-munger**. Each filter, constraint or munger must have been defined before being referenced in **def-sensor**, and they must be of the appropriate sense type. Example:

```
(def-sensor nose smell nose1 (combine-smells (smell-distance-constraint 5.0))
  ())
```

Combine-smells is assumed to be some filter which combines the input list of smells into one smell. Smell-distance-constraint is a constraint which only allows smells within a given distance to be sensed. For the nose1 sensor that distance is always 5.0 meters. Nose1 has no field mungers.

When more than one entity-filter is specified, the list of entities is successively subjected to each of the filters in turn. The constraints will be tried before the filters since they are usually cheaper.

(sense '<sensor-name>)

[1]<object> is normally part of the organism's body, although it doesn't have to be!

Gets the list of sensory data (entities) appropriate to *<sensor-name>*, passes it through the appropriate <entity-filters> and <field-mungers>, and returns the processed list of senses. For example, *(sense 'nose1)* would get all the smells which pass through the **combine-smells** filter.

(def-filter <name> <params> <sense-type> . <body>)

General sense-filter defining macro. Defines a method for filtering out entities from a given list of entities.

<name> is the name given to the filtering function.

<params> is a list of the names of the parameters given to the function, () if none.

<sense-type> is one of the legal sense types. Only entities corresponding to the sense-type will be passed through the filter.

<body> is the actual code to do the filtering. The syntax ". <body>" means that it is an implicit progn, as with **defun**. The resulting function that is written will contain the <body> as the body, and will take the parameters described in *<params>*, plus two parameters, *entities* and *sensorobj*, as input. *Entities* is a list of entities of the appropriate type. *sensorobj* is the object that has sensed the entities. The <body> should return a list of entities which is a subset of the input list. Example:

```
;;; This simple filter reduces the list of objects to a list containing
;;; the N closest objects to the organism.  It is assumed that my-min is
;;; defined to be like min but take a :key and :lowest param.
(def-filter closest-objects (n) vision
  (my-min entities :lowest n
          :key #'(lambda (obj) (get-object-distance obj sensorobj))))
```

(def-field-munger <name> <params> <sense-type> <field> . <body>)

Associates a munger with a field of entities of the type corresponding to *<sense-type>*. Whenever a field named *<field>* is retrieved, it is munged by each of the field-mungers associated with it by **def-field-munger**. For now, fields are only munger *after* the filters have been executed; i.e., the filters will be looking at the original values of the fields, but the organism will be seeing the munged values. When the body is executed it can be assumed that *field* is bound to the value of the field. For example:

```
;;; For an organism which can only see red or blue.  It is
;;; assumed that color-sum is some fn which returns the "color sum".
(def-field-munger make-red-or-blue (thresh) vision outside-color
  (if (< (color-sum field) thresh)
      (make-color :red 255 :blue 0 :green 0)
      (make-color :red 0 :blue 255 :green 0)))
```

(def-constraint <name> <params> <sense-type> . <body>)

Defines a constraint for the entities of type *<sense-type>*, and calls it *<name>*. The parameters are the same as for <def-filter>. The difference is that constraints are given one entity at a time, instead of a list of entites. For each entity they are given, they return either nil or non-nil, to indicate whether the entity should be allowed to stay in the overall list.

<params> is a list of the names of the parameters which are given to the constraint function.

<body> is the progn to compute whether the given entity should stay. The body will have as parameters the parameters described in the *<params>* list, as well as the two parameters *entity* and *sensorobj*.

Although filters can do everything constraints can, constraints may only take one entity at a time, and are provided because they may be implemented more efficiently. Whereas filters operate on a whole list of entities, if an entity can be removed from consideration by only looking at it and possibly the sensor, then it should be expressed as a constraint. In fact, most filtering can be expressed as constraints. For instance:

```
;;; Only sense the smells that are near your "nose" (i.e., less than
;;; max-distance meters away).
(def-constraint smell-distance-constraint (max-distance) smell
  (< (vector-distance (emission-origin entity)
                      (object-location sensorobj))
     max-distance))
```

(pp-sensor <sensor-name>) Pretty-prints the given named sensor. <sensor-name> does not have to be quoted.

2.1. Auxiliary Sensor Functions

The obvious sense functions such as see and smell can be built on top of the basic *sense* command. For example, *see* means to get all the data from all the sensors of type *vision*. However, there is as yet no general model for integrating the various sensory data coming from the different sensors, so the different lists of entities sensed by the different sensors is simply returned as a list of lists.

2.1.1. See

Return a list of all the entity lists returned by calling *sense* on each of the sensors of type *vision*.

2.1.2. Smell

Return a list of all the entity lists returned by calling *sense* on each of the sensors of type *smell*.

2.1.3. Taste

Return a list of all the entity lists returned by calling *sense* on each of the sensors of type *taste*.

2.1.4. Hear

Return a list of all the entity lists returned by calling *sense* on each of the sensors of type *hearing*. NOTE: be sure not to call listen instead, since it is a reservered Commonlisp function!

2.1.5. Feel

Return a list of all the entity lists returned by calling *sense* on each of the sensors of type *touch*.

2.2. Object-getting Functions

2.2.1. get-object <object-number>

Returns the object with the given number. If the object had already been loaded on the current cycle, then it doesn't bother to re-copy the object from C.

2.2.2. get-object-named <name>

Returns the object with the given name. Name must be a string. Uses get-object.

3. Effector Libraries

Effector functions allow the organism to make changes to the environment in a "fair" way. Fair is defined here as: try to conserve the laws of conservation of momentum and energy. They may also be thought of macro-operators, the primitive operators being the ones provided by the LispSEI.

Effector functions should be written by individual members of World Modelers since everybody has a different idea as to how different effector functions should work. However, a few sample effectors are provided in the World Modelers SEI directory.

I. The Files

The SEI library module is in /usr/worldm/src/sei/seidef.lisp. The compiled form of this for Perqs is seidef.sfasl; for Suns it is seidef.lbin. An initial filter and effector library is located in /usr/worldm/src/sei/seilib.lisp, seilib.sfasl and seilib.lbin. Additions of general use may be made to this library, or users may make their own libraries. Libraries are compilable. This document is /usr/pshell/worldm/sei/seiprop.mss.